



ESCOLA DE MATEMÁTICA APLICADA

---

## Aula Prática 5 - ALN - Fatoração QR

Aluno: *Gustavo Murilo Cavalcante Carvalho*

E-mail: [gustavomurilo012@gmail.com](mailto:gustavomurilo012@gmail.com)

Programa: *Bacharelado em Matemática Aplicada*

Disciplina: *Álgebra Linear Numérica*

Professor: *Antonio Carlos Saraiva Branco*

---

Data: *15 de junho de 2024*

---

## Sumário

Problema 1 .....	1
Implementação .....	1
Testes .....	1
Problema 2 .....	4
Implementação .....	4
Testes .....	4
Problema 3 .....	6
Implementação .....	6
Testes .....	6
Problema 4 .....	7
Implementação .....	7
Testes .....	9
Problema 5 .....	10
Implementação .....	10
Testes .....	10

## Problema 1

O método de Gram Schmidt para ortogonalização de vetores é um método iterativo que, dado um conjunto de vetores linearmente independentes, gera um conjunto de vetores ortogonais. O método é baseado na projeção de um vetor sobre os vetores anteriores.

## Implementação

### qr\_GS.m

```
% Entradas:
% A - matriz (m x n)
% Saídas:
% Q = matriz (m x n) ortogonal
% R = matriz (n x n) triangular superior
function [Q,R] = qr_GS(A)
    [m, n] = size(A);

    % Inicializa matrizes
    Q = zeros(m,n);
    R = zeros(n);

    for j = 1 : n
        v = A(:,j); % j-ésima coluna de A

        % Obtém, por Gram-Schmidt, v o j-ésimo vetor de uma base ortogonal
        for i = 1 : j-1
            R(i,j) = dot(Q(:,i), A(:,j));
            v = v - R(i,j) * Q(:,i);
        end

        R(j,j) = norm(v,2);
        Q(:,j) = v / R(j,j); % j-ésimo vetor de uma base ortonormal
    end
end
```

## Testes

A seguir estão algumas matrizes selecionadas para testar as funções implementadas neste trabalho.

$$A = \begin{bmatrix} 1 & 2 & 2 \\ -1 & 1 & 2 \\ -1 & 0 & 1 \\ 1 & 1 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 2 & 1 & 2 & 4 \\ 3 & 1 & 2 & 2 \end{bmatrix}$$

A é uma matriz ideal para ortogonalização (tanto que é um exemplo dado pelo Poole), pois contém vetores para uma base de um subespaço vetorial.

$B$  é uma matriz mágica de ordem par, portanto é muito má condicionada (ela é quase singular), o que é interessante para os testes.

$C$  é uma matriz retangular com mais colunas do que linhas, o que não é o caso esperado de acordo com a apresentação da fatoração QR (a que é feita no Poole), contudo é um cenário de prova para a implementação e interpretação do método.

**Testando para o cenário ideal (a matriz  $A$ ), temos:**

```
>> [QCa, RCa] = qr_GS(A);

QCa =
    0.5    0.67082   -0.40825
   -0.5    0.67082         0
   -0.5    0.22361    0.40825
    0.5    0.22361    0.8165

RCa =
     2         1         0.5
     0    2.2361    3.3541
     0         0    1.2247
```

Considerando  $A = QR$ , para verificar a ortogonalidade de  $Q$ , calculamos  $Q^T Q$  e para verificar a acurácia decomposição  $QR$ , calculamos  $QR - A$ .

```
>> QCa' * QCa
     1         0         0
     0         1   -2.7e-17
     0   -2.7e-17         1

>> QCa * RCa - A
     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

Pode ser visto que a decomposição  $QR$  obtida foi muito boa.  $Q$  não é por muito pouco (o erro é irrelevante, tem grandeza  $10^{-17}$ ) a identidade, e a multiplicação de  $Q$  e  $R$  resulta em  $A$ .

**Para as outras matrizes ( $B$  e  $C$ ), temos:**

Ambas são boas fatorações, afinal a multiplicação das matrizes resulta na matriz original, ou algo muito próximo disso. Vale ressaltar que a fatoração funciona para  $C$  (uma matriz com mais colunas do que linhas), mesmo que esvaziada do sentido.

```
>> QCb*RCb - B
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0

>> QCc*RCc - C
-2.2e-16    0    0    0
    0    0    0    0
```

Testando a ortogonalidade de B, vemos que o resultado não é o melhor, muitas entradas são muito próximas de zero, outras nem tanto (na ordem de  $10^{-1}$ ). Isso é uma consequência do mal condicionamento de B.

```
>> QCb'*QCb
    1 -2.7e-17 4.9e-16 0.55125
-2.7e-17    1 -5.5e-16 -0.25841
4.9e-16 -5.5e-16    1 -0.7925
0.55125 -0.25841 -0.7925    1
```

Para C, temos que avaliar algo diferente, afinal a matriz Q associada a ela não pode ser ortogonal, ela não é LI. Note que há um bloco que é a identidade, o que acontece devido ao fato de que os dois primeiros vetores são LI, se fossem outros, o bloco da identidade estaria em outra posição (ao menos é).

```
>> QCc'*QCc
    1 1.4e-15 0.00377 0.00377
1.4e-15    1 -0.99999 -0.99999
0.00377 -0.99999    1    1
0.00377 -0.99999    1    1

>> QCc(1:2, 1:2)'*QCc(1:2, 1:2)
    1 1.4e-15
1.4e-15    1
```

## Problema 2

A versão modificado tem como objetivo trazer uma melhor estabilidade numérica ao método de Gram-Schmidt. A ideia é que, ao invés de calcular a projeção de um vetor sobre os outros já ortogonalizados, calculamos a projeção do vetor mais atualizados sobre os já ortogonalizados. Isso evita a acumulação de erros numéricos.

## Implementação

### qr\_GSM.m

```
% Entradas:
%   A - matriz (m x n)
% Saídas:
%   Q = matriz (m x n) ortogonal
%   R = matriz (n x n) triangular superior
function [Q,R] = qr_GSM(A)
    [m, n] = size(A);

    % Inicializa matrizes
    Q = zeros(m,n);
    R = zeros(n);

    for j = 1 : n
        v = A(:,j); % j-ésima coluna de A

        % Obtém, por Gram-Schmidt, v o j-ésimo vetor de uma base ortogonal
        for i = 1 : j-1
            R(i,j) = dot(Q(:,i), v); % Passa a usar o vetor mais atualizado
            v = v - R(i,j) * Q(:,i);
        end

        R(j,j) = norm(v,2);
        Q(:,j) = v / R(j,j); % j-ésimo vetor de uma base ortonormal
    end
end
```

## Testes

Para a matriz A, essa modificação obteve os mesmos resultados que a anterior. Portanto, não há necessidade expor os resultados.

Para a matriz B, uma matriz má condicionada, a modificação alterou minimamente uma entrada da matriz R. Quanto à matriz Q, notamos que a última coluna dessa nova Q é diferente da anterior.

```
>> QCb(:,4)' % Quarta coluna da matriz Q obtida com qr_GSC
    0.32233    0.40291    0.64466   -0.56408

>> [QMb, RMb] = qr_GSM(B);

>> QMb(:,4)'
    0.94679    0.063119    0.25248   -0.18936
```

Associada a essa mudança, vemos uma melhor ortogonalidade da matriz Q utilizando o método de Gram-Schmidt modificado. Contudo, devido à dimensão da matriz esse ganho é mínimo.

Considerando matrizes mágicas de ordem par (caracterizadas por serem muito mal condicionadas), fica claro que esse algoritmo produz uma matriz Q muito mais próxima da ortogonalidade.

```
>> [Q ~] = qr_GS(magic(20));
>> [QM ~] = qr_GSM(magic(20));
>> norm(Q'*Q - eye(20))
    16.989
>> norm(QM'*QM - eye(20))
    0.99972

>> [Q ~] = qr_GS(magic(100));
>> [QM ~] = qr_GSM(magic(100));
>> norm(Q'*Q - eye(100))
    97
>> norm(QM'*QM - eye(100))
    0.99995
```

Resta testar a matriz C. Agora, com essa modificação, deixamos de obter uma fatoração eficaz. Ambas as matrizes possuem entradas NaN (não faz sentido verificar ortogonalidade ou acurácia). Isso se deve ao fato de C possuir colunas LD (linearmente dependentes). Logo essa modificação do método restringe a função a matrizes LI (linearmente independentes).

```
>> [QMc, RMc] = qr_GSM(C)

QMc =
    0.5547    0.83205    NaN    NaN
    0.83205   -0.5547    NaN    NaN

RMc
    3.6056    1.3868    2.7735    3.8829
         0    0.27735    0.5547    2.2188
         0         0         0         NaN
         0         0         0         NaN
```

## Problema 3

### Implementação

qr\_GSP.m

```
% Código do problema 3
```

### Testes

## Problema 4

### Implementação

#### qr\_House.m

```
% Entradas:
%   A - matriz de entrada (m x n)
% Saídas:
%   U - matriz (m x m) contendo os vetores normais
%   R - matriz (m x n) triangular superior
function [U, R] = qr_House(A)
    [m, n] = size(A);

    % Inicializar a matriz U com zeros
    U = zeros(m, m);

    for i = 1 : n
        % Extrair a coluna atual a partir da i-ésima linha até o final
        x = A(i:m, i);

        % Obtém o vetor normal ao hiperplano de reflexão
        if x(1) > 0
            x(1) = x(1) + norm(x, 2);
        else
            x(1) = x(1) - norm(x, 2);
        end

        u = x / norm(x, 2); % Normaliza o vetor
        U(i:m, i) = u; % Armazena o vetor em U

        % Aplica a transformação de Householder à submatriz de A
        A(i:m, i:n) = A(i:m, i:n) - 2*u*(u'*A(i:m, i:n));
    end

    R = triu(A); % Os valores abaixo da diagonal seriam próximos de 0
end
```



**qr\_House\_min.m**

```
% Entradas:
% A - matriz (m x n)
% Saídas:
% U - matriz (m x n) contendo os vetores normais
% R - matriz (m x k) triangular superior
function [U,R] = qr_House_min(A)
    [m, n] = size(A);

    % Determina a dimensão correta
    if m == n
        k = m - 1;
    else
        k = min(m,n); % Essa alteração abrange o caso onde m < n
    end

    % Inicializa matrizes
    R = A;
    U = zeros(m, k);

    for i = 1 : min(m,n)
        x = A(i:m, i);

        % Obtém o vetor normal ao hiperplano de reflexão
        if x(1) > 0
            x(1) = x(1) + norm(x, 2);
        else
            x(1) = x(1) - norm(x, 2);
        end

        u = x / norm(x,2); % Normaliza o vetor
        U(i:m, i) = u;     % Armazena o vetor em U

        % Aplica a transformação de Householder à submatriz de A
        A(i:m, i:n) = A(i:m, i:n) - 2*u*(u'*A(i:m, i:n));
    end

    R = triu(A(1:k, 1:n)); % Para que coincida com
end %endfunction
```

**constroi\_Q.m**

```
% Entradas:
%   U - matriz (m x n) com vetores de Householder
% Saídas:
%   Q = matriz (m x n) ortogonal
function [Q] = constroi_Q(U)
    % Obtém dimensões de U e inicializa Q
    [m,n] = size(U);
    Q = eye(m,n);

    for i = 1 : n
        u = U(:,i);

        % Aplica a transformação de Householder pela direita Q*(H - u*u')
        Q = Q - 2*Q*(u*u');
    end
end
```

**Testes**

## Problema 5

### Implementação

#### espectro.m

```
% Entradas:
%   A - matriz (n x n)
% Saídas:
%   S = vetor (n x 1) ortogonal
function [S] = espectro(A, tol)
    % Definição de variáveis
    erro = tol + 1;
    S = diag(A);

    while tol <= erro
        % Processo iterativo
        [Q, R] = qr_GSM(A);
        A = R * Q;

        % Verificação de convergência
        novo_S = diag(A);
        erro = norm(S - novo_S, 'inf');

        S = novo_S; % Atualiza o resultado
    end
end
```

### Testes

Para os testes, gero matrizes com números inteiros uniformemente distribuídos entre 1 e 9. A matriz é então multiplicada por sua transposta para que seja simétrica e, portanto, tenha autovalores reais. Então comparamos os autovalores obtidos pela função criada com os autovalores obtidos pela função eig do MATLAB.

```
>> M = randi(9,5,5);

>> M = M' * M;

>> flip(eig(M))
    606.58    61.75    29.118    4.9587    0.58848

>> S = espectro(M, 1e-12)
    606.58    61.75    29.118    4.9587    0.58848
```

Os testes acima (assim como todos os posteriores) foram feitos com uma tolerância de  $10^{-12}$ , e mesmo assim foram obtidos muito rapidamente.

Uma ideia interessante para escalar os testes para matrizes maiores, é verificar a norma entre a diferença do resultado das funções, ao invés de comparar os vetores diretamente.

```
>> N = randi(9,10,10);
>> N = N' * N;
>> S = espectro(N, 1e-12);
>> S - flip(eig(N))
    9.0949e-13
   -8.5265e-14
    1.4211e-14
   -1.4211e-14
    5.6843e-14
    3.5527e-14
   -1.3603e-11
    1.3443e-11
   -3.1974e-14
    1.9054e-13

>> norm(espectro(N,1e-12) - flip(eig(N)))
    1.9148e-11
```

Verificando para uma matriz de ordem 100, temos:

```
>> O = randi(9,200,200)
>> O = O' * O;
>> norm(espectro(O,1e-12) - flip(eig(O)))
    7.5866e-10
```

Esse resultado foi bem preciso, mas a função já demorou bem mais para convergir (aproximadamente 1 minuto).

Para matrizes maiores (de ordem 300 por exemplo), a nossa função passa a ser muito lenta (demorando quase 6 minutos para convergir) o que torna o seu uso inviável.